**HW-**

**Section A -** **Recursion:**

1. **Difference between Direct and Indirect Recursion**
   * **Direct Recursion:** A function calls itself directly within its body.
   * **Indirect Recursion:** A function calls another function, which in turn calls the original function.

Example of Direct Recursion:

void directRecursion(int n) {

if (n > 0) {

System.out.println(n);

directRecursion(n - 1);

}

}

Example of Indirect Recursion:

void functionA(int n) {

if (n > 0) {

System.out.println(n);

functionB(n - 1);

}

}

void functionB(int n) {

if (n > 0) {

System.out.println(n);

functionA(n - 1);

}

}

1. **What is Tail Recursion? How is it optimized?**
   * **Tail recursion** occurs when the recursive call is the last operation in the function before returning.
   * It is optimized by the compiler using **Tail Call Optimization (TCO)**, which replaces the recursive call with a loop, reducing the call stack usage.

Example:

void tailRecursion(int n) {

if (n == 0) return;

System.out.println(n);

tailRecursion(n - 1); // Last operation is the recursive call

}

**Optimization:** In languages like JavaScript and some functional languages, TCO replaces recursive calls with iteration to prevent stack overflow.

1. **How does recursion use the call stack?**
   * Each recursive call is pushed onto the **call stack**, storing its execution context.
   * When the base case is reached, the calls are popped from the stack in **Last In, First Out (LIFO)** order.

Example:

void recursionExample(int n) {

if (n == 0) return;

System.out.println(n);

recursionExample(n - 1);

System.out.println(n); // This executes after recursion unwinds

}

**Execution for recursionExample(3):**

* + recursionExample(3) -> Prints 3, calls recursionExample(2)
  + recursionExample(2) -> Prints 2, calls recursionExample(1)
  + recursionExample(1) -> Prints 1, calls recursionExample(0) (base case reached)
  + recursionExample(1) resumes -> Prints 1
  + recursionExample(2) resumes -> Prints 2
  + recursionExample(3) resumes -> Prints 3

1. **How to convert a recursive function to an iterative function?**
   * Use a **loop** (e.g., for, while) instead of recursion.
   * Use an **explicit stack (if necessary)** to simulate recursion.

Example (Factorial Calculation):

**Recursive Approach:**

int factorialRecursive(int n) {

if (n == 0 || n == 1) return 1;

return n \* factorialRecursive(n - 1);

}

**Iterative Approach:**

int factorialIterative(int n) {

int result = 1;

for (int i = 1; i <= n; i++) {

result \*= i;

}

return result;

}

1. **Recursive function to find the Greatest Common Divisor (GCD) of two numbers**
   * Uses **Euclidean Algorithm**, which states that GCD(a, b) = GCD(b, a % b).

**Recursive Approach:**

int gcdRecursive(int a, int b) {

if (b == 0) return a;

return gcdRecursive(b, a % b);

}

Example Call:

System.out.println(gcdRecursive(48, 18)); // Output: 6

**Section B** - **Searching Techniques:**

1. **Different Searching Algorithms:** Searching algorithms are used to find an element within a data structure. Some commonly used searching algorithms include:
   * **Linear Search**
   * **Binary Search**
   * **Jump Search**
   * **Interpolation Search**
   * **Exponential Search**
   * **Fibonacci Search**
   * **Ternary Search**
2. **Linear Search and Its Time Complexity:**
   * Linear search is a simple searching algorithm that checks every element in a list sequentially until the target element is found or the list ends.
   * **Algorithm:**
     1. Start from the first element.
     2. Compare each element with the target.
     3. If a match is found, return the index.
     4. If no match is found, return -1.
   * **Time Complexity:**
     1. **Best Case:** O(1) (if the element is found at the beginning)
     2. **Worst Case:** O(n) (if the element is at the end or not present)
     3. **Average Case:** O(n)
3. **Binary Search and Its Time Complexity:**
   * Binary search is an efficient algorithm that works on sorted arrays by repeatedly dividing the search space in half.
   * **Algorithm:**
     1. Find the middle element of the array.
     2. If the middle element is the target, return its index.
     3. If the target is smaller, search the left half; otherwise, search the right half.
     4. Repeat the process until the element is found or the search space is empty.
   * **Time Complexity:**
     1. **Best Case:** O(1) (if the middle element is the target)
     2. **Worst Case:** O(log n)
     3. **Average Case:** O(log n)
4. **How Binary Search Works on a Sorted Array:**
   * Binary search requires a sorted array.
   * It starts by identifying the middle element and comparing it with the target.
   * If the middle element is the target, the search ends.
   * If the target is smaller, the search continues in the left half; if larger, it continues in the right half.
   * This process repeats until the element is found or the search space is exhausted.
5. **Advantages and Disadvantages of Binary Search:** **Advantages:**
   * Much faster than linear search, especially for large datasets.
   * Time complexity is O(log n), making it efficient.
   * Requires fewer comparisons compared to linear search.

**Disadvantages:**

* + Works only on sorted arrays.
  + Requires extra processing to maintain a sorted list.
  + More complex to implement compared to linear search.

**Section C -** **Backtracking:**

1. What is backtracking, and how does it work?

Backtracking is a general algorithmic technique that is used to solve problems incrementally by building candidates to a solution step by step and removing those that fail to satisfy the constraints of the problem. It systematically explores all possible solutions in a recursive manner and backtracks (goes back) when a certain condition is not met. This approach is useful for solving combinatorial and constraint-satisfaction problems.

Backtracking works by:

* Choosing a possible solution step (candidate)
* Checking if it satisfies the given constraints
* If it does, moving forward with the next step
* If it does not, undoing the last step (backtracking) and trying a different approach
* Repeating this process until a solution is found or all possibilities are exhausted

1. What are the key components of a backtracking algorithm?

The key components of a backtracking algorithm include:

* **Choice**: Selecting possible candidates or options for the solution.
* **Constraints**: Defining rules that the solution must satisfy.
* **Goal**: Establishing the condition that determines when a valid solution is found.
* **Backtracking Mechanism**: The process of undoing choices when constraints are violated and exploring alternative paths.
* **Recursive Exploration**: Using recursion to navigate through different possibilities efficiently.

1. What are some real-world applications of backtracking?

Backtracking is used in various real-world scenarios, including:

* **Sudoku Solver**: Finding valid number placements in a Sudoku puzzle.
* **N-Queens Problem**: Placing N queens on an NxN chessboard such that no two queens attack each other.
* **Graph Coloring**: Assigning colors to a graph’s vertices such that no two adjacent vertices share the same color.
* **Word Search Puzzle**: Searching for words in a grid by navigating adjacent cells.
* **Pathfinding (Maze Solving)**: Finding paths in mazes using recursive backtracking.
* **Combinatorial Optimization**: Generating and testing permutations, subsets, and combinations.
* **Regular Expression Matching**: Implementing pattern matching algorithms.

1. Explore and write the psudocode for the Sudoku Solver using backtracking.

function solveSudoku(board):

if no empty cell in board:

return True

(row, col) = findEmptyCell(board)

for num from 1 to 9:

if isValid(board, row, col, num):

board[row][col] = num

if solveSudoku(board):

return True

board[row][col] = 0 // Backtrack

return False

function isValid(board, row, col, num):

for i from 0 to 8:

if board[row][i] == num or board[i][col] == num:

return False

startRow = (row / 3) \* 3

startCol = (col / 3) \* 3

for i from 0 to 2:

for j from 0 to 2:

if board[startRow + i][startCol + j] == num:

return False

return True

function findEmptyCell(board):

for i from 0 to 8:

for j from 0 to 8:

if board[i][j] == 0:

return (i, j)

return None

1. Explore and write the logic/pseudocode for the N-Queens problem using backtracking.

function solveNQueens(board, row, N):

if row == N:

printSolution(board)

return

for col from 0 to N-1:

if isSafe(board, row, col, N):

board[row][col] = 1

solveNQueens(board, row + 1, N)

board[row][col] = 0 // Backtrack

function isSafe(board, row, col, N):

for i from 0 to row:

if board[i][col] == 1:

return False

for i, j from (row, col) to (0, 0) diagonally left:

if board[i][j] == 1:

return False

for i, j from (row, col) to (0, N-1) diagonally right:

if board[i][j] == 1:

return False

return True

function printSolution(board):

for row in board:

print(row)